BCIT

COMP 7005 - Computer Network and Protocols

Data Communications Principles

Final Project

A01059056 - Alex Laroche

A00953335 - Kalen Tara

Purpose:

The main purpose of this project is to design and implement a **Send-And-Wait** protocol. The protocol will be based around the half duplex system while also using sliding windows. This system will send multiple packets between the host and the client on a LAN network with unreliable connections between the server and the client. Our design requires us to discard random packets to mimic the noise of an unreliable network.

Essentially the project can be broken into three devices. A transmitter, a receiver and a noise generating unreliable network emulator. This unreliable network emulator will mirror a certain bit error rate. This will be done using three physical machines over a LAN connection.

Our design is based on taking a UDP connection and converting it into a connection that acts like a TCP connection. This paper will walk you through our design and the implementation that we followed.

Constraints:

This project had no language limitations, but we elected to use python. This design is entirely limited to the application layer protocol. Essentially we are required to use a UDP like connection while also handling network errors. As mentioned above, the network emulator will act as an unreliable network with random errors being created. The methods of the emulator will be following command line prompts. There are three different source modules: a client, a server and the network emulator.

Although it is not required, we will treat the IP address as specified from a singular configuration file. In addition, the ports will be specified in this file.

Technical Objectives:

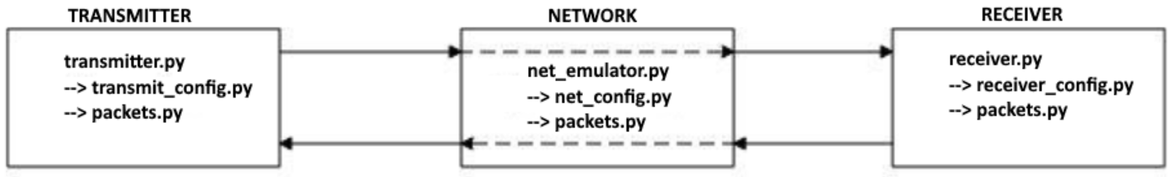
The objectives of this paper and project are to design and implement an application layer protocol built on the UDP system while acting like a TCP system. This will be done by having three source modules and three physical machines. Our source modules are the conceptual network module, the transmission module, and the receiver module. All of these modules will have their own configurations, although they will all rely on a central packet logic configuration file. Each of these modules will have specified ports to follow for their transmissions.

Our conceptual network module design will include features that act like timeouts, packet drops and other aspects of noise. The conceptual network module will emulate an unreliable network where the packets are being sent over. Although this project will be running over a LAN, the emulator will act similar to transfers done over the internet. Using randomized functions in our code, the conceptual network module will randomly drop packets as well as ACKs in order to create a specific bit error rate. Ultimately the network emulator should follow commands sent through the command line or even hard coded into the module itself.

Our transmitter and receiver will function entirely on the UDP system, while acting as a TCP system with aspects like the three-way handshake.

Application Structure and Usage:

File Structure



Each box above signifies a specific machine. Each file inside the box is what the machine requires to function properly, they all require their distinctive files, their configs, and packets.py which is a general file holding the basic functions which are used by all the machines. In the configuration file of each machine the addresses and ports of all machines must be specified or a connection will not be made.

Usage

Configuration

Each machine has its own configuration file, the usage is as followed:

Basic configuration:

- **net_address:** The ip address in which the Network emulator is being hosted.
- **net_port:** The port in which the Network emulator is being hosted.
- **transmit_address:** The ip address in which the Transmitter is being hosted.
- **transmit_port:** The port in which the Transmitter is being hosted.
- **receive_address:** The ip address in which the Receiver is being hosted.
- **receive_port:** The port in which the Receiver is being hosted.

Network config:

- **average_length:** The average length of time which to wait before sending the data to either the Transmitter or Receiver.

Transmitter config:

- **window_size:** The maximum window size of the transaction. To function properly, values must be 2^*(1,2,4,8...etc).

timeout: The length of time the transmitter will wait until it retransmits the packets.
max_packet_size: The maximum length of data which can be held in a specific packet.

Application

Ensure configuration of all corresponding config files are properly set up. Begin the network emulator(net_emulator.py) and enter the desired bit error percentage prompted in the command line. Then start the Receiver (receiver.py). Then start the Transmitter (transmitter.py), the connection will begin with the SOT sent indicated by the packet_type: “S”. The Receiver will then respond with an ACK(“.”) and the connection will have been established, the transmitter can begin sending messages to the receiver. The user will be prompted with a message and send it to the receiver. When the user is done sending messages to the receiver, on in the transmitter console enter a Keyboard Interrupt (Ctrl + C on windows) and the Transmitter will send the EOT finishing the connection and printing extra statistics to the log files.

Packet Structure:

Packet Object Structure:
{
src_address: string
src_port: int
dst_address: string
dst_port: int
packet_type: string
seq_num: int
ack_num: int
data: string
window_size: int
}
src_address: Source Address indicates the address of the sender.
src_port: Source Port indicates the port which the sender uses.
dst_address: Destination Address indicates the address where the sender is sending the packet
dst_port: Destination Port indicates the port where the sender is sending the packet
packet_type:
S = Start of Transmission
P. = Push ACKs (Data being Sent)
. = ACK, Data has been received
E = End of Transmission
seq_num: The Sequence Number of the packet

ack_num: The Acknowledgement Number of the packet
data: The separated data string which the packet holds
window_size: The amount of packets sent in that transmission

Pseudo Code:

net_emulator.py

function start_connection:

Initializes a server and binds it to the network address and port specified in the configuration file.

The User is then prompted a required input where they must specify the Bit Rate Error Percentage.

While there is a connection, listen for packets.

When the packets arrive, determine if the packet is dropped (Specified by Bit Error Rate)

If the packet is lost, simply print the lost packet and return to listening for packets

If the packet is NOT lost, then send the Receiver message to the Transmitter or send the
Transmitter message to the Receiver.

receiver.py

function create_packet:

Returns a packet Object with the specific required entries

function receive_data:

Listens for packets; PUSH ACKs

If packet is out of order, ADD 1 to duplicate ack

Increment the the Acknowledgement number by the Sequence number and the length of data

Create an ACK packet (".") and send the packet to the network emulator.

If the Packet Type is "E" then raise an exception which logs the Total sent ACKs over the transmission.

function start_connection:

Initializes a server and binds it to the network address and port specified in the configuration file.

While the receiver is listening, Allow Packets to be Received.

If the Packet Type is "E" then raise an exception which logs the Total sent ACKs over the transmission.

transmitter.py

function create_packet:

Returns a packet Object with the specific required entries

function transmit:

While loops is created to loop through the provided data

Buffer Sequence and Ack Numbers are set in case of Retransmission

Fixes the length of the data to ensure it can full the current Window Size

Loops through the current Window Size, and sends packets accordingly

Sets A buffer for the Expected Ack

Calls the Window Ack Function

Empties the Sent Data Buffer

Unless the Socket is an SOT, then increase the Window Size = Window Size * 2, until it reaches the Window Size specified in the config

function send_data:

Create an PUSH ACK packet ("P.") using the provided data, and send the packet to the network emulator.

Setting the Sequence Number to Sequence Number + the length of the Packet Data

Appends the PUSH ACK to the Buffer List used for Retransmission

function receive_ack:

Set a timeout for the socket's listening to indicate the need for a Retransmission.

Receive ACK and set the Sequence Number and ACK Number.

If the socket timeouts then set that the order is False and Retransmitting will begin.

function retransmit_packets:

Sets The Order to True

Resend the packets from the Saved Buffer

function window_ack:

Receives ACKs based on the current Window Size.

If the order is broken then break out of the Loop set the Sequence Number and ACK to their tx buffer counterparts,copy the expected acks from the buffer.

Call on the retransmit_packets function.

Recall the window_ack.

function split_data:

Splits data into the maximum packet size specified in the config

function start_connection:

Initializes a server and binds it to the network address and port specified in the configuration file.

Sends an SOT to indicate that the transmitter will start sending Data

While there is a connection, have the user input data to send, call the function split_data on the Data Entered, and the call the function transmit to transmit the data

If a Keyboard Interrupt Occurs, Transmit the EOT, Receive the ACK, then Exit the script

Environment:

The environment for this project was set up with three physical computers. All of these systems were all on the same local network. Our original tests were done using a single computer and then using virtual machines. However, when we decided to show our real demonstrations of the functionality of our project, we decided it was best to use our physical computers.

The transmitter was done from a windows 10 environment using a wired ethernet cable. As you can see in our tests, all of the transmissions were sent from a windows terminal command line.

The receiver was Kalen's personal ubuntu server. This server is a physical box set up with a wired ethernet cable. For the file system, the file location on both the receiver and the network were under a file named '7005final'. This was the easiest way to make it so all transfers, including the SCP's afterwards, were not to be misplaced.

Finally the network emulator was set up on Kalen's fedora drive. This was the only part of the system that used wireless connection over ethernet. The drive itself was plugged into a laptop, which made wireless the obvious solution.

Conclusion:

conclusion is gay? - kalen